NASA Contractor Report 177999

ICASE REPORT NO. 85-45

NASA-CR-177999
19860004446

ICASE

PARALLEL COMPUTATION WITH THE FORCE

Harry F. Jordan

NAS1-17070, AFOSR 85-1089

October 1985

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton Virginia 23665

Parallel Computation with the Force†

Harry F Jordan
Department of Electrical and Computer Engineering
University of Colorado

Abstract

A methodology, called the force, supports the construction of programs to be executed in parallel by a force of processes The number of processes in the force is unspecified, but potentially very large The force idea is embodied in a set of macros which produce multiprocessor Fortran code and has been studied on two shared memory multiprocessors of fairly different character The method has simplified the writing of highly parallel programs within a limited class of parallel algorithms and is being extended to cover a broader class This paper deals with the individual parallel constructs which comprise the force methodology Of central concern are their semantics, implementation on different architectures and performance implications

†Research was supported in part by NASA Contract No NAS1-17070 and by the Air Force Office of Scientific Research under Grant No AFOSR 85-1089 while the author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665

Conceptual Basis for the Force

The force [1] methodology for parallel programming arose in trying to produce high performance parallel programs in a shared-memory multiprocessor running up to 200 processes on the same user program [2] Multiprogramming was not an issue, and all emphasis was on single problem solution speed Partly for performance measurement purposes and partly for program manageability, a programming style emerged in which a single piece of code was written which could be executed by a force of processes in parallel The number of processes constituting the force is constant during execution but is bound as late as the beginning of execution, and may be one

The force technique insulates the programmer from all process management and leaves him the issues involving process synchronization Since processes are established by a program independent driver at the beginning of execution time, parallelism is introduced at the top of the procedure hierarchy This has the effect of insulating the user from parallelism issues with results similar to those obtained by encapsulating parallelism below a particular level in the procedure hierarchy The study of techniques for using the force in a program is essentially a study of synchronization mechanisms which are independent of the number and identities of the processes synchronized

Several advantages arise out of independence from the number of processes It is not necessary to design algorithms with a detailed dependence on the, potentially very large, number of processes executing them The choice of the optimal number of processes can be made at run time on the basis of system hardware configuration and load Since complete independence from the number of processes implies correct execution with only one process, the issues of arithmetic correctness and multi-process synchronization can be separated in the testing of a program

Statements written in a force program are implicitly executed by all processes in parallel Variables appearing in statements are divided into local variables, having separate instances for each process, and global variables, shared among all processes of the force An assignment statement, for example, may combine the values of global and local variables to produce a local or global result If the result is local, no assignment conflict is possible If it is global, then assignment conflict must be prevented, either by allocation of disjoint sections of a global data structure to multiple processes or by synchronizing the assignment across processes, say by enclosing it in a critical section or by using producer/consumer synchronization on the variable assigned Library or user subroutines which are either free of side effects or carefully synchronized can be invoked in parallel, one copy for each process

One way in which disjoint sections of a global data structure, specifically an array, may be allocated to multiple processes is to schedule distinct index values in a DOALL across processes Index values may either be assigned statically to processes once the number of processes is known, in which case we speak of a prescheduled DOALL, or processes may dynamically schedule themselves by obtaining distinct values of a global index variable as they become available to execute the loop body, known as a self-scheduled

DOALL In either case, the body of the DOALL is executed once for each index value by some one of the processes The choice of prescheduling or self-scheduling may impact performance as a result of uneven workload division or of conflict on access to the global index variable

The programming language associated with the force consists of some simple extensions to the Fortran language, which are currently implemented as macros which are expanded by a language independent preprocessor The target Fortran system must, of course, include ways of creating multiple processes and of supporting synchronized access to global variables A currently operational set of macros produces Fortran for the HEP computer [3], built by Denelcor, Inc , and a set is being constructed for the Flex/32 [1], built by Flexible Computer Corporation The macros interact through the variables of a parallel environment, which contains some general information such as the number of processes and some machine dependent items

Parallelism Constructs of the Force

The macros currently constituting the force can be divided into several classes, as shown in Fig 1 The first class deals with parallel program structure The macros *Force* and *Forcesub* respectively begin parallel main programs and parallel subroutines They make the parallel environment variables available to the macros within that program module as well as making the number of processes and a unique identifier for the current process available to the user at run time An *End Declarations* macro marks the beginning of executable code and provides target locations for declarations and start up code which may be generated by the macros A *Join* macro terminates the parallel main program It is the last statement executed by all processes of the force

Macros of the second class deal with variable declaration This class currently includes only *Global* and *Local* macros Global variables are associated with Fortran common while local variables are ordinary Fortran variables local to a separately compiled program module Sharing of local variables among several program modules, but local to one process, can only be accomplished by parameter passing The static allocation flavor of Fortran makes it difficult to build a structure of common variables with one instance for each process when the number of processes is not known until execution time

Macros of another class distribute work across processes The most familiar construct is the DOALL, which is employed when instances of a loop body for different index values are independent and can thus be executed in any order Two versions are provided The *Presched DO* divides index values among processes in a fixed manner which depends only on the index range and the number of processes The *Selfsched DO* allows processes to schedule themselves over index values by obtaining the next available value of a shared index as they become free to do work For situations in which it is desirable to parallelize over both indices of a doubly nested loop, both prescheduled, *Pre2DO*, and self scheduled, *Self2DO*, macros are available

```
Macros associated with program structure
    Force <name> of <# procs> ident <proc #>
        <declarations>
    End declarations
        <force program>
    Join

    Forcesub <name> of <#procs> ident <proc #>
        <declarations>
    End header
        <subroutine body>
    RETURN

    Forcecall <name>(<parameters>)

Declaration macros
    Global <variable names>
    Local <Fortran declaration>

Macros specifying parallel execution
    Pcase on <variable>
        <code block>
    Usect
        <code block>

    End pcase

    [Pre|self]sched DO <n> <var> = <i1>, <i2>, <i3>
        <loop body>
<n>  End [pre|self]sched DO

Synchronizing macros
    Barrier
        <code block>
    End barrier

    Critical S<variable>
        <code block>
    End critical

    Produce <variable> = <expression>          (producer)
      =    Use(<variable>                      (consumer)
```

Figure 1 Specific Macros for a Force Program

Independence of the loop body instances over both indices is, of course, required for correct operation A similar construct is the parallel case, *Pcase,*

which distributes different single stream code blocks over the processes of the force Execution conditions can be associated with each block, and any number of these conditions may be true simultaneously No order of evaluation of the conditions is specified, and each will be evaluated by one arbitrarily selected process Thus conditions depending only on global variables are most meaningful

At the heart of the force methodology are the synchronization macros They characterize the approach to parallel programming and provide the means for controlling the force so that coherent and deterministic computation can be performed Two subclasses of synchronization are control flow oriented synchronizations and data oriented synchronizations The key control oriented synchronization is the barrier since it provides control of the entire force Its semantics are that all processes must execute a *Barrier* macro before one arbitrarily chosen process executes the code block between *Barrier* and *End Barrier* When the code block is complete, the entire force begins execution at the statement following the *End Barrier* Although all but one process are temporarily suspended by a barrier, no process termination or creation takes place and all local process states are preserved across the barrier Operations which depend on the past computation, or determine the future progress, of the entire force are typically enclosed in a barrier

Another control based synchronization is the critical section, familiar from the operating systems literature Statements between *Critical <variable>* and *End Critical* may only be executed by one process of the force at a time This mutual exclusion extends to any other critical section with the same associated variable Data oriented synchronization is provided by the elementary producer-consumer mechanism, in which global variables have a binary state, full or empty, as well as a value Execution by some process of the macro, *Produce <variable> = <expression>*, waits for the variable to be in the empty state, sets its value to that of the expression and makes it full, all in a manner which is atomic with respect to the progress of any other process Similarly, the macro, *Use(<variable>)*, appearing in an expression returns the value of the variable when it becomes full and sets it empty Variables in the wrong state may cause these macros to block the progress of a process Auxiliary macros for full/empty variables are *Purge <variable>*, which sets a variable empty regardless of its previous state, and *Copy(<variable>)*, which waits for the variable to be full and returns its value but does not empty it

A major weakness in the current set of force macros is that it does not smoothly support decomposition of a program into parallel components on the basis of functionality The *Pcase* macro offers the rudiments of this, but only allows one process to execute each of the parallel functions What is desired is a macro, *Resolve*, which will resolve the force into components executing different parallel code sections The section of code for each component would start with *Component <name> strength <number>*, which would name the component and specify the fraction of the force to be devoted to this component The component strengths would be estimated by the programmer on the basis of any knowledge available about the

computational complexity of each component A macro, *Unify,* would reunite the components into a single force The implementation of *Resolve* is complicated by the conflicting demands of generality and efficiency If the number of components is larger than the number of processes in the force, then inter-component synchronization may deadlock unless the components are co-scheduled over the available processes An implementation which produces process rescheduling at every possible deadlock point and is still efficient when the number of processes exceeds the number of components is under development

Incorporation of a *Resolve* macro would make it useful to extend the barrier idea A barrier should be able to specify whether only the processes in the current component are to be blocked or whether all processes in the parent force are to participate In the case of recursively nested *Resolve* constructs, the barrier might specify a nesting level relative to the one in which it appears

The *Resolve* idea promises a mechanism for functional decomposition of programs into parallel components, but there is one more capability of parallel programming environments with explicit process management which is not addressed by the force This is the ability to give away work to "available" processes in a dynamic manner during execution This ability is most called for by tree algorithms and dynamic divide-and-conquer methods It would be desirable for the force to contain a mechanism for efficiently handling such algorithms without making the user responsible for explicit process management or losing the benefits of independence of the number of processes A mechanism related to resolve might be applied at each tree node but could lead to much process management overhead in cases where the correct thing to do is merely to traverse a subtree with the one remaining process

Interrelationships Between the Primitives

The semantics of the parallelism constructs in the force imply certain restrictions on the way they are used together in a program Several of the constructs restrict execution to a single stream within some code block *Barrier* and *Pcase* limit execution of enclosed blocks to a single process while critical section code is eventually executed by all processes, but only one at a time Thus constructs which depend on multiple, simultaneous execution, such as DOALL, *Pcase* or *Barrier* should not appear within such blocks A critical section within a *Barrier* is meaningless, but critical sections have definite use within two or more code blocks of a *Pcase* construct Nested critical sections have meaning when the associated locking variables are different Data oriented synchronization primitives may occur within singly executed code without restriction, other than the natural possibility of deadlock In fact, initialization of full/empty variables is usually done within a singly executed block

Parallel loops do not restrict the execution of their bodies to a single process, but they do limit execution of the body for each index value to one process Thus constructs which depend on full parallel execution cannot appear

within DOALLs These include *Barrier, Pcase* and other DOALLs The inconsistency in the parallelism requirements of nested DOALLs is the reason for supplying multiple index DOALLs for parallel execution of loop bodies which are independent over the Cartesian product of two or more index sets Critical sections, Produce and Use are quite useful within DOALLs and often lead to programs in which the distributed nature of synchronization reduces its effect on program performance

Subroutine invocation within a force program can be done either with a *Forcecall* or an ordinary Fortran CALL Only the *Forcecall* makes the parallel environment available to the subroutine called Since a force subroutine invoked by *Forcecall* assumes that all processes of the force will enter it, a *Forcecall* must not appear within a code body in which parallel execution has been restricted Thus, Forcecalls are not meaningful within Barrier, Pcase, Critical or DOALL constructs An ordinary CALL implies execution of a subroutine in single stream on behalf of one or more processes Since any Fortran based parallel system must support multiple independent execution of subroutines, such as those in the mathematical library, subroutines must have separate local variable states for all processes executing them An ordinary Fortran subroutine or function call may thus appear within any code section of a force program The subroutines or functions so invoked contain no parallel constructs and access by them to any shared variables must be controlled externally if it is desired

The *Resolve* construct is intended to produce a new parallel execution environment within each of its components, differing from the original only in the number of processes Thus all of the parallelism primitives have meaning within a force component The implementations of the primitives must, of course, refer to the parallel environment of the component rather than of the original force The meaning of Barrier, as has been noted, can be extended to refer to higher levels of a nested component structure, but it retains its original meaning with respect to the immediate component with no modification of its semantics Barrier, Pcase and the DOALLs have an action limited to the component in which they appear Critical sections and data oriented synchronizations can synchronize operations within the current component with operations in any other components which share the corresponding variables

Performance Issues

Various features of the force methodology are related to the performance of a parallel computer system An overall principle used in selection of primitive operations for inclusion in the force was that the semantics of each primitive should be simple enough to admit of an efficient implementation across the range of shared memory multiprocessors The simple process model, consisting of program counter, local variables and unique identifying index, also contributes to low overhead implementation on most shared memory machines Process priorities and parent-child relationships, for example, can significantly complicate the implementation of a parallel programming

system on some multiprocessors which do not directly support such features

The primitive operations of the force define a virtual machine, and the generality of this machine yields independence from the details of the underlying hardware This benefit of machine independence and portability need not, however, suppress all machine performance issues at the level of force programming Pratt [5] points out that a virtual machine for parallel execution should make "visible," as programming alternatives, distinctions which may reflect major hardware performance differences The clearest example of such alternatives within the force is the existence of both a prescheduled and a self-scheduled DOALL

At the level of the abstract machine, the process interactions implied by pre- and self- scheduling are different Prescheduling since it allocates index values to processes in a fixed way as soon as the number of processes is determined, will split the workload evenly across processes only if processors run at similar speeds and the amount of computation specified by the DOALL body is independent of index value On the other hand, no process interaction is required to allocate the index values, each process can determine its own portion of the work independently In contrast, the self-scheduling technique allows processes to load balance at execution time by obtaining further index values whenever they complete the work connected with previous values This is done at the expense of a short critical section to obtain, increment and store a shared index variable

For a given underlying hardware, these distinctions at the abstract machine level can be translated into performance differences by using a few general characteristics of the hardware system The most important parameters for the pre- versus self- scheduling comparison are the size, in execution time, of a minimal critical section to access and update a shared index and the number of processes competing for this access When combined with the program dependent parameters of the mean and standard deviation of the DOALL body size over the set of index values they allow a determination of which type of scheduling will lead to better performance

Implementation Issues

Implementation issues can be addressed on the basis of variations in the two current implementations Several hardware differences between the HEP and the Flex/32 multiprocessors influence implementation of the force macros A minor, but basic level, difference is that all memory in the HEP can be shared by all processes so only Fortran variable scope issues are involved in implementing global variables In the Flex/32, only a restricted portion of the address space is accessible by processes running on different processors so shared variables must physically reside in these addresses as well as satisfying Fortran conventions for name sharing by different modules The shared address space on the Flex/32 is large enough and its access time near enough to that of local memory that this should not be an issue except for programs with very large global data requirements

The basic synchronization mechanism in HEP is the locking full/empty bit in each memory cell Locks in the Flex/32 are separate and, although there are 8192 of them, they form a scarcer resource than HEP synchronization elements Furthermore, since the HEP has hardware to support the temporary suspension of processes, the user can do synchronizations directly while the manipulation of locks in the Flex/32 must be done through the operating system Figure 2 shows critical sections for both machines and notes the user instruction versus system call distinction The Flex/32 Con-Current C system supports the association of a lock with any shared variable to which synchronized access is made, so at this level the machine differences are not major, as far as implementation of the force macros is concerned As shown in Fig 2, the critical section macro has an associated variable to allow for distinct sets of interacting critical sections In both implementations this becomes a global variable which is locked (directly in the HEP and via system call in the Flex/32) on entry to and unlocked on exit from the critical section

The *Produce* and *Use* macros are quite different on the two systems simply because they correspond directly to single memory access instructions on the HEP and must involve the locks on the Flex/32 Implementation of

HEP

Critical lock1	call awrite(lock1, true)
<code block>	<code block>
End Critical	call laread(lock1)

Flex/32

Critical lock1	call CClock(1, "lock1")
<code block>	<code block>
End Critical	call CCunlck(1, "lock1")

Single instruction HEP Fortran intrinsics		Flex/32 operating system calls	
awrite	wait for empty, write, set full	CClock	wait for unlocked and lock
laread	wait for full, read, set empty (logical)	CCunlck	unlock

Figure 2 Implementation of Critical Sections

Produce and *Use* for fewer than 8192 variables might be done using the hardware locks on the Flex/32, but full/empty access to individual elements of a large array requires a software supported association of variables with full/empty bits Synchronized access to the bits and their associated variables needs to employ a combination of the locks and events supported by the hardware

Implementation of the *Barrier* macro shows some clear differences between the two systems In both cases, the semantics requiring all processes to arrive before the code section is executed is supported by a shared counter synchronized as in the critical section Two barrier mechanisms have been used on the HEP In systems small enough that memory contention is not a problem, the last process to increment the shared counter executes the code section and fills a memory location which the other processes are attempting to read Processes must then count down the counter as they exit the barrier, with the last one resetting the lock If memory contention is a problem, the ability to control processes at the user level allows writing of a HEP assembly language routine in which all but the last process to enter a barrier terminate execution to be recreated with their previous state by the last process to enter the barrier In the Flex/32, process control is a system function The system, however, supports the concept of shared events, connected to processes in a broadcast configuration Here, processes entering the barrier wait on the event, except for the last one, which executes the code block of the barrier and then activates the event Verifying that each process connected to the event has seen it is part of the operating system support, so no exit code is required The first mechanism for the HEP is contrasted with the Flex/32 implementation in Fig 3

Conclusions

The design of a parallel programming system involves a combination of the issues of utility with those of implementation efficiency The utility issues have been treated in previous papers [1] [6] while this work concentrates on the individual macro semantics and implementation issues The force methodology supports efficient implementation by the simplicity of its process model and lack of complex semantics in individual parallel constructs At least two multiprocessors with shared memory admit of straightforward implementations

HEP

```
                                  if (lwaitf(ilock)) continue
Barrier                           nloc = iaread(nbar) + 1
                                  call awrite(nbar, nloc)
                                  if (nloc eq np) then

  <code block>                      <code block>

                                   call sete(ilock)
                                   call awrite(olock, true )
                                  endif
                                  if (lwaitf(olock)) continue
End Barrier                       nloc = iaread(nbar) - 1
                                  call iawrite(nbar, nloc)
                                  if (nloc eq np) then
                                   call sete(olock)
                                   call awrite(ilock, true )
                                  endif
```

Flex/32

```
                                  call CClock(1, "nbar")
                                  nloc = nbar + 1
Barrier                           nbar = nloc
                                  call CCunlck(1, "nbar")
                                  if (nloc eq np) then

  <code block>                      <code block>

                                   call CCactev(1, 4, "bar")
End Barrier                       else
                                   call CCwev(1, 4, "bar")
                                  endif
```

HEP single instruction intrinsics		Flex/32 operating system calls	
waitf	- wait for full, read	CClock	- wait free, set lock
aread	- wait full, read, set empty	CCunlck	- clear lock
awrite	- wait empty, write, set full	CCactev	- activate event
sete	- set empty	CCwev	- wait for event

Figure 3 Implementation of Barriers

REFERENCES

[1] H F Jordan, "Structuring parallel algorithms in an MIMD, shared memory environment," *Proc 18th Hawaii Int'nl Conf on Systems Sciences,* Vol. II, pp 30-38 (1985), to appear in *Parallel Computing,* 1985

[2] H F Jordan, "HEP architecture, programming and performance," in *Parallel MIMD Computation: The HEP Supercomputer and its Applications,* J S Kowalik, Ed., MIT Press (1985)

[3] M C Gilliland, B J Smith and W Calvert, "HEP - A semaphore-synchronized multiprocessor with central control," *Proc 1976 Summer Computer Simulation Conf,* Washington D C, pp 57-62 (July 1976).

[4] *Flex/32 Multicomputer: System Overview,* Flexible Computer Corporation, Dallas, Texas (1985)

[5] T W. Pratt, "Pisces An environment for parallel scientific computation," *ICASE Rept No. 85-12,* NASA Langley Research Center, Hampton, VA, NASA CR-172544 (February 1985).

[6] N R Patel and H F. Jordan, "A parallelized point rowwise successive over-relaxation method on a multiprocessor" *Parallel Computing,* Vol 1, No 3&4 (December 1984).

1 Report No	2 Government Accession No	3 Recipient's Catalog No
NASA CR-177999 ICASE Report No. 85-45		

4 Title and Subtitle	5 Report Date
PARALLEL COMPUTATION WITH THE FORCE	October 1985
	6 Performing Organization Code

7 Author(s)	8 Performing Organization Report No
Harry F. Jordan	85-45
	10 Work Unit No

9 Performing Organization Name and Address	11 Contract or Grant No
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665	NAS1-17070, AFOSR 85-1089
	13 Type of Report and Period Covered Contractor Report
12 Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546	14 Sponsoring Agency Code 505-31-83-01

15 Supplementary Notes

Langley Technical Monitor:
J. C. South Jr.
Final Report

Submitted to Proc. 1986 Intl. Conf. on Parallel Processing

16 Abstract

A methodology, called the force, supports the construction of programs to be executed in parallel by a force of processes. The number of processes in the force is unspecified, but potentially very large. The force idea is embodied in a set of macros which produce multiprocessor Fortran code and has been studied on two shared memory multiprocessors of fairly different character. The method has simplified the writing of highly parallel programs within a limited class of parallel algorithms and is being extended to cover a broader class. This paper deals with the individual parallel constructs which comprise the force methodology. Of central concern are their semantics, implementation on different architectures and performance implications.

17 Key Words (Suggested by Author(s))	18 Distribution Statement
multiprocessor programming environment parallel programs	61 - Computer Programming & Software Unclassified - Unlimited

19 Security Classif (of this report)	20 Security Classif (of this page)	21 No of Pages	22 Price
Unclassified	Unclassified	13	A02

End of Document